Monte Carlo Methods and the Computers of the Future

Malvin H. Kalos[1]

Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

Ultracomputer Note #83

April, 1985

1. Introduction

Speaking as a would-be user of computers of the future, I find that they seem bewildering. My instinct is to ask for a VAX that runs ten million times faster than my 780 with perhaps a thousand times as much memory. Our salesman is not offering any of those just yet. So, I have to come to this conference hoping for helpful, specific direction from the assembled experts. Instead I am surrounded by conflicting advice. It seems as though I will have to learn to choose intelligently among (at least) the following alternatives as far as architectures are concerned:

SIMD vs. MIMD organization
Many small vs. fewer capable processors operating in parallel
Distributed memory vs. shared memory
Central control vs. distributed control
Many von-Neumann vs. data driven or demand driven
Lattice communication vs. optimal logarithmic network
Fixed communication vs. reconfigurable communication

As for languages, I learn from different speakers that

(1) FORTRAN is acceptable if lightly modified;

(2) FORTRAN is acceptable for low-level procedures providing the high-level specification is given graphically;

(3) FORTRAN is acceptable for low-level procedures providing the high-level specification is written in GIBBS;

---

[1] Invited talk at Conference on Algorithms, Architectures, and Computers of the Future held at the University of Texas at Austin, March, 1985. This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract number DE-AC0276ER03077.

(4) I must start to use an applicative language to exploit all the parallelism and the sooner I can junk FORTRAN the better.

Finally, I learn that to pick my way through these interconnected mazes, that is how to choose algorithms and architectures that together will be well suited for my applications, it would be well to await the emergence of the future expert systems in which the counsel I need will be embedded. The temptation is strong to retire and become a computer scientist. Why try to do computations, when, for less work and more pay, one can simply talk about them?

In any case, it will be necessary to absorb the algorithmic implications of different proposed architectures and relate those in turn to various applications. Many important applications programs are not dominated by single algorithms and demand architectures that can gracefully handle different algorithms with different requirements for granularity, parallelism, communication patterns, and the other parameters that may characterize them.

To do this in anything like complete generality will be a formidable task. It will take years of study and synthesis by researchers from many disciplines. This talk aims at a modest beginning for the problems which are commonly attacked by Monte Carlo.

2. The Two Dimensional Ising Problem

Let me begin by describing in a little detail what is undoubtedly the simplest useful Monte Carlo problem and algorithms, those for treating two-dimensional Ising models. The problem is easy to describe: consider a square in which a square lattice is put down. Every point of the lattice has a local variable, σ, which can have one of two values, "up" and "down", or + 1 and -1. In the figure + 1 is represented by a white castle and -1 by a black. Black and white squares show interpenetrating square sublattices.

The physics of the problem is defined by the energy E_k of the k^{th} site

$$E_K = -J\sum_n \sigma_k \sigma_n \qquad (1)$$

where n goes over the four nearest neighbors of site k. Thus E_k measures the

degree to which neighboring spins of the system are aligned. In the figure a particular spin is shown with its neighbors: its energy is -2 J. The full energy of the system is

$$E = \frac{1}{2}\sum_k E_k = -\frac{1}{2}J\sum_k\sum_n \sigma_k \sigma_n \tag{2}$$

Now equilibrium statistical mechanics tells us that all arrangements of all spins $X = \{\sigma_1,\sigma_2, \cdots \sigma_n\}$ are possible, but that the chance that a given X is observed is random with probability given by the Boltzmann distribution:

$$p(X) = p(\sigma_1,\sigma_2, \cdots \sigma_n) = \frac{\exp[-E(X)/k_BT]}{\sum_X \exp[-E(X)/k_BT]} \tag{3}$$

where k_B is Boltzmann's constant and T is the absolute temperature. Problems in equilibrium statistical mechanics can be answered simply by computing averages of some observable A(X) with respect to p(X):

$$\langle A\rangle = \sum_X A(X)p(X) = \frac{\sum_X A(X)\exp[-E(X)/k_BT]}{\sum_X \exp[-E(X)/k_BT]} \tag{4}$$

If one wants to evaluate $\langle A\rangle$, the sum might be done explicitly, a formidable task. For a modest grid of 100 × 100 there are $2^{10^4} \simeq 10^{3010}$ possible configurations. Direct enumeration, even at a rate of 10^{10} per second, would take about 10^{2992} years.

We know, however, from the theory of Monte Carlo methods that accurate answers can be obtained from a sample chosen at random. In particular, if a set of configurations $X_1,X_2, \cdots X_m$ is sampled at random from p(X), and the arithmetic average formed

$$A_M = \frac{1}{M}\sum_{m=1}^{M} A(X_m), \tag{5}$$

then A_M will be a numerical approximate to $\langle A\rangle$. It is difficult to say in advance how big M must be for a prescribed accuracy, but vastly fewer than 2^N are required. Often some thousands to millions of samples suffice.

Unfortunately, the sampling problem is itself very difficult, or would be except for a remarkable algorithm due to Metropolis et. al. [1] In its simplest form applied here, the following steps are required:

(1) Sample k at random from 1 to N.

(2) Set σ'_k to $-\sigma_k$

(3) Calculate $\Delta E_k = -2E_k$

(4) With probability, $\min[1,\exp(-\Delta E_k/k_BT)]$, set $\sigma_k = \sigma'_k$ i.e., "accept" the flipped spin; else set σ_k as the value (i.e. "reject" the flipped spin)

When these are repeated many times the distribution of X converges to the Boltzmann distribution. Evidently the successive x_m are highly correlated. The estimates of A_M have lower statistical accuracy than if the x_m were independent. The degree of correlation depends on the problem; it is very large at low temperature and near the critical point.

As described, the algorithm is sequential. A trivial (but effective) parallel version would simply assign separate but independent replications of the whole system to run on different processors, which are coupled only to the extent that the averages A_M (or many such averages) and estimates of their errors are eventually tallied in common memory locations.

It is possible to prove, however, that a mathematically correct alternative is to flip at the same time all spins not directly coupled by the energy function. That is all spins on white squares in our diagram can be flipped together, followed by all spins on black squares. This leads to highly efficient vector algorithms; two recent realizations are given in references [2] and [3]. The second of these is interesting in that all aspects of the algorithm, flipping, testing, acceptance/rejection are carried out in parallel on all bits of a word using completely vectorizable boolean operations.

If one imagines building a special purpose parallel computer for the Ising problem, one can visualize assigning a processor to treat one spin or a rectangular block of spins. It seems self-evident that the optimal architecture will require only nearest neighbor connections; in other words the processors should be connected as a square lattice. The "physics" seems to demand it.

Whether this is true depends on what one means by "optimal". The possibilities include:

(1) Optimal for the hardware engineer. Nearest neighbor communications architecture is easiest to build.

(2) Optimal for the language designer. Selecting one among four connections for message passing saves fuss and bother.

(3) Optimal for this algorithm. Nothing else is needed for the naive Metropolis algorithm.

(4) Optimal for the problem.

If one tries seriously to optimize for the problem, life becomes more difficult. It becomes necessary to think. Is there a better algorithm? What is the optimal algorithm? On very general grounds [4] it is possible to state that all existing realizations of the Metropolis algorithm are non-optimal in the sense that there always exists an alternative with faster decaying sequential correlations. These alternatives are characterized by the need for more global information about the system, e.g. they might use block spins or spin wave characterizations of the system. In particular, Schmidt [4] has given an algorithm for the one-dimensional Ising model in which it takes 2N

steps to sample a configuration that is independent of all others. It is better than Metropolis by an arbitrary ratio at low temperatures. A significant feature of the algorithm is that spins are sampled which will be separated in the final configurations by 2, 4, 8,...sites. Thus nearest neighbor connections are not enough for efficient data communication.

The Metropolis algorithm may be regarded as a time driven simulation in which one or many spins are flipped in response to a synchronous clock. It is possible also to construct an event-driven simulation [5] in which a next accepted flip is chosen and a corresponding time delay is calculated. Again this will be very effective at low temperatures. The algorithmic character of this method dependent upon arrays of pointers that are manipulated at random. Thus it does not lend itself to vectorization. Parallel versions of it are straightforward.

We see that if one talks about optimal architectures for the Ising model problem it is necessary to look beyond the properties of the *Hamiltonian*. In a certain sense the physics does determine the architecture, but the knowledge required involves the qualitative character of the *solution*.

3. Transport Monte Carlo

Let us now discuss another widely used Monte Carlo method, one used to calculate the passage of radiation and particles through matter. It has the appearance of a simulation of the natural stochastic processes that govern the underlying interactions. That means that frequent data dependent branches appear in the code. Some of these are choices among the data that describe the interactions, some the (often very complex) description of the geometry of the medium, and many are the random choices that arise in a numerical stochastic process. Often the choices lead to different subprograms. Thus the program flow is that of small granularity and high logical complexity. This is not at all what usually comes to the minds of computer scientists when they think of "numerical calculations."

Another complication is the appearance of branching (i.e., "birth and death" processes), either because they appear in the basis physics as fission, (n,2n) or other multiplying reactions, or because variance reduction methods that call for branching have been used. This increases the logical complexity of the program and, along with the natural – sometimes very large – variance of the time for completion of a process owing to the fluctuations in the number of particle collisions, poses a significant load balancing issue for parallel computation.

Monte Carlo methods are used because of their numerical efficiency in treating many-dimensional problems with fully accurate descriptions of the distributions of fluxes in time, energy and direction. Perhaps even more important, they permit *flexible* and *accurate* treatment of cross sections and all other physics, geometry of media, detector geometry and physics, and of

fluctuation phenomena. Finally, they permit objective estimation of computational error. Briefly put, transport Monte Carlo is used to give greatest fidelity to nature, to experiments, and to engineering requirements.

Future computers intended for use for these methods must preserve and enhance their advantages.

What then are the architectural requirements for this class of Monte Carlo methods? They can be vectorized, but only at the expense of extensive and deep reprogramming. Even worse, with present and easily foreseeable programming languages, such vectorized codes lose the great flexibility inherent in the method.

The methods lend themselves rather easily to MIMD parallel treatment. The required reprogramming is straightforward and confined to high levels of the program organization.

Not all MIMD architectures will be equally facile, however. For very large problems, especially involving neutron transport, very large data and "answer" bases must be available for essentially random access by all participating processors. I estimate that some tens of millions of words will be needed: 10^5 words/nuclide to describe neutron interactions, and perhaps as many as 50 nuclides; 10^5 words to describe the geometry of the medium; 10^7 words to record fluxes and their errors. These arrays are much too large to be replicated more than a few times. Random access to the data is required, with relative frequencies to different parts that are problem dependent. Obviously one wants as much speed as can be effectively used. As has been indicated, significant capability for load balancing will be needed. Finally, the architecture should be such as to retain the flexibility and portability of existing Monte Carlo programs.

In my opinion, these requirements point almost unambiguously towards shared memory architectures.

4. Algorithmic Characterizations Of Monte Carlo Methods

We may examine other Monte Carlo Algorithms in a similar spirit. A brief list of salient characters for some algorithms is the following:

Small Quadrature

> Repeated sampling on about 10 or more dimensions.
> Individual calculations may be large; i.e. granularity may be large.
> May be adaptive; global results may affect the parameters or character of algorithms as sampling proceeds.

Transport Monte Carlo

Resembles a simulation of a natural stochastic process.
Random walk in up to 7 dimensions.
Branching (Birth/Death) possible.
Low granularity.
Decisions highly data dependent.
Large data bases may be needed.

Spin Monte Carlo

Random walk in discrete space (lattice with + 1 or 1)
$10^2 - 10^7$ sites or more.
Granularity low.
Program flow may be very regular or not.

Classical Fluid Monte Carlo

Random walk in $10^2 - 10^5$ dim.
Granularity large.

Quantum Monte Carlo (Including Green's Function Monte Carlo)

Random walk in $10^1 - 10^4$ dim.
Granularity large to very large.
May have branching.
Walks may interact.

To summarize, different Monte Carlo algorithms differ in the following ways:

- Dimensionality
- Granularity of decision making
- Regularity of logical flow
- Presence of branching
- Character of dependence of random walks
- Regularity of communication
- Pattern of communication

The different algorithmic natures influence choice of computing architecture. It seems likely that truly optimal architecture may differ from one application to another. That tends to support the impulse to build unique, initially experimental machines. Such activities are fully justified when it is clear that a specific algorithm can be used over the useful life of several years of such a device, that it can be maintained, and that the kind of programming environment normally present in a general purpose machine is not necessary.

On the other hand, we see that some Monte Carlo algorithms demand the full power of general purpose machines. In addition, the exploration of future applications and the search for more nearly optimal algorithms for present applications are best served by the most powerful, most general

architectures that can be devised. It should not be forgot that our knowledge of methods, Monte Carlo and otherwise, rests on the all-purpose uniprocessors of the past. We must lay the groundwork for the problems of the future and the algorithms of the future by designing the computers of the future. Let us look now at one candidate for such computers.

5. The NYU Ultracomputer

This is an architecture in which very many conventional processors are tightly coupled through a shared memory to cooperate effectively on major computations. Its design is predicated upon the idea that microcomputers will become more powerful and cost effective; on projected existence of 32 bit microprocessors comprising one or a very few chips, capable of 10-20 MIPS performance, and on positing the wide availability of megabit memory chips. Its design also reflects the assumption that VLSI design of a few special purpose chips is feasible, and that these too will be cheap when manufactured in quantity. In these regards, the regular small part-number design of the Ultracomputer contrasts with the more conventional type of supercomputer design involving many different VLSI chips with irregular interconnection patterns. There is good reason to believe that the cost of reaching a given supercomputer performance level can be minimized by using large numbers of each of very few chip types organized in as systematic and homogeneous a design as possible.

Given this view, the question arises of how the most effective organization of processors and memory can be attained. One idea is to distribute complete processors with memory on some kind of network, i.e. to associate a portion of memory directly with each processor. Such an approach has engineering advantages and can be used very effectively if it can be assumed that almost all the data treated by every processor comes from the local memory unit. However, such a clean data partitioning is sometimes impossible to impose, so that an architecture in which memory is shared globally between processors is likely to be more easily programmable for a wide variety of applications. Whether a truly effective version of such a shared memory architecture can be built then becomes the crucial question. It is clear that an ideal machine, in which every processor can access any memory location in one processor clock cycle, cannot be constructed because of limitations of local connectivity (i.e., of electronic "fan in"). Instead memory access requires a latency time logarithmic in the number of processors. Nevertheless, we contend that the current design of the NYU Ultracomputer resolves this central difficulty successfully and is a feasible and effective approximation to an idealized parallel computer. To buttress this claim, the next few paragraphs present the most salient characteristics of the Ultracomputer design.

5.1. Architecture

A simple idea of the Ultracomputer's organization can be obtained from the accompanying figure.

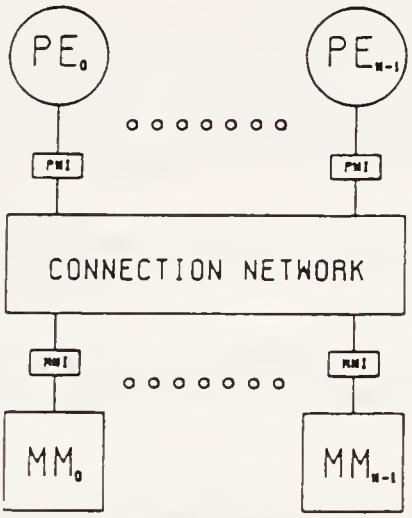

Fig. 1. Block diagram.

It shows an array of individual processors or "PEs" an equal number of memory modules "MMs" (containing, say, several megabytes each), and an interconnection network that permits any processor to access any MM. For this crucial task of interconnection, the Ultracomputer uses a significantly modified Omega or Banyan [6] network.

Two components of secondary importance are also included: a memory-network interface (MNI) at each MM and a processor-network interface (PNI) at each PE. Besides the usual functions of reformatting information from or to the form required by the network, the MNI and PNI have certain novel features described below.

The design for the network results in a memory latency under light loads about equal to the time for a few short arithmetic instructions. This makes use of a cache essential, but, as supported by extensive simulation, latency is not so large as to preclude the effective use of shared data in the array of memory modules. Prefetching of data, preferably as directed by an optimizing compiler, can be used to ameliorate the effects of latency. Processor multiprogramming will be very effective also.

5.1.1. The Processors: The Fetch and Add Coordination Primitive Each elementary processor in the Ultracomputer will be a standard microprocessor with 32 bit arithmetic including fast floating-point operations. A novel instruction called "Fetch and Add" is also provided. The effect of this instruction is as follows: the processor computes an increment *e* and an address *addr* in shared memory. The operation

F&A(addr,e)

returns the previous contents of *addr* to the processor, and replaces [*addr*], the contents of *addr*, by its previous value increased by *e*. The operation of the Ultracomputer as a whole is such that if two processors simultaneously issue an F&A for the same address *addr*, then the outcome is as if they occurred in some serial order. That is, if processor i issues F&A(addr,e_1) and processor j issues F&A(addr,e_2) and [*addr*] was originally A, then either

Processor i receives A and Processor j receives A+e_1
(as though F&A(addr,e_1) had occurred first),

or

Processor i receives A+e_2 and Processor j receives A
(as though F&A(addr,e_2) had occurred first).

In either case, [*addr*] becomes A+e_1+e_2 at the end. The case of three or more simultaneous F&A instructions directed at the same location works in an analogous way. All this is important since our work has shown that (a) any number of simultaneous F&A operations can be handled by properly designed switching hardware in a highly efficient way, and (b) all necessary synchronization in a machine of the proposed type can be programmed efficiently using F&A as the only nonconventional instruction.

The complete hardware implementation of F&A involves operations distributed among the PNIs, network switches, and MNIs, as described below.

5.1.2. The Network Communication in the Ultracomputer makes use of the so-called Omega Network, an asymptotically optimal communication net developed in the mid-1950's by Bell Telephone Laboratories. An Omega network connecting 2^d inputs with 2^d outputs requires d stages each containing 2^{d-1} 2 by 2 switches[2]. The figure shows an example of an Omega network that connects eight inputs with eight outputs. This network topology has three properties that make it suitable for use in the Ultracomputer. First, switching is very simple and does not require precomputation of a path: at each stage a different bit of the memory module address is used to select which of two possible output directions is taken by a message moving through the network. A second property is that the network can be viewed as a superposition of binary trees: each processor is the root of a tree whose 2^d leaves are the memory modules and each memory module is the root of a tree whose 2^d leaves are the processors. Finally, there is a unique path between a given processor and any memory module so that two packets of

[2]As in the more general "Banyan" networks, it is possible to use m by n switches. To

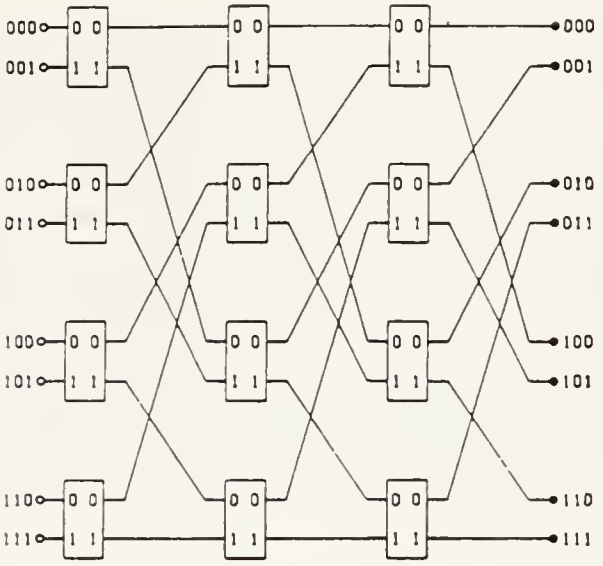

Fig. 2. Omega-network ($N = 8$).

information directed simultaneously at the same address by two processors must intersect at a switch.

In building very large processor ensembles, it is essential that the maximum total throughput of the network (i.e., its "bandwidth") be proportional to the number of processors. To ensure that this holds asymptotically, as the number of stages grows, two conditions are required of the network. One is that it must be message switched. That means that requests for data from memory are inserted into the network by the PNI as short messages that travel from switch to switch. Because they can be issued every switch cycle time, such messages are pipelined to memory and back in our design[3].

Since conflicts will occur between messages that must pass along intersecting paths, a second requirement on the network is that messages that cannot move forward because of conflicts be either combined with other messages, or stored at a switch for later transmission. That is, the "switches" constituting the Ultracomputer's processor-to-memory network must have some memory and some elementary processing power.

5.1.3. The Switches To meet the requirement for large bandwidth, the Ultracomputer network switches must be able to recognize messages aimed at the same address. If both are fetches, then only one fetch is transmitted toward memory but the identity of both transmitting processors is saved.

---

simplify the exposition, we restrict attention to 2 by 2.

[3]The alternative, (a "circuit switched" network) in which a path is cleared from processor to memory and kept open while information flows both ways, will suffer from an increasing probability of interference as the network increases in size.

When the fetched information returns to the switch in question, it is transmitted toward both processors. Conflicting stores are treated by suppressing one of them but acknowledging both.

The treatment given to concurrent Fetch and Add instructions targeted at the same address is significant. Suppose a switch receives F&A(addr,e_1) and F&A(addr,e_2). It will save e_1, and transmit a combined message, namely F&A(addr,e_1+e_2). When the previous contents of *addr* arrives back at the switch, it is transmitted back toward the processor that issued F&A(addr,e_1) and [*addr*]+e_1 is transmitted toward the other. Concurrent messages having identical addresses are detected by the output queues present in the switch, which are based on the VLSI systolic design of Guibas and Liang [7], but enhanced to support address matching in parallel with the standard queue functions [8].

Precisely because a binary tree exists between all the processors and any single memory module, when any number of Fetch and Add instructions are issued simultaneously for the same address, they can combine in pairs at successive stages of the network while converging toward addr, and split into different results on the way back. Thus any number of Fetch and Adds targeted at the same address can execute in the same time as a single one. This key property allows us to construct bottleneck free system software accomplishing the coordination of very many closely coupled processes.

5.1.4. Processor Network Interface The PNI module connecting each processor to its corresponding network port includes a cache containing instructions and private data for the process active in its PE. This improves performance in two important ways: cached information can be accessed without the latency time of the network, and the load on the network is decreased. We avoid depending on any breakthrough in the problems associated with consistency among multiple caches, by not caching shared read-write data[4].

5.1.5. Memory Network Interface The MNI includes an adder so that it can carry out the final addition associated with an F&A(a,e). In performing this operation, the MNI retrieves [*a*] and transmits it back toward the requesting processor, adds *e*, and stores the sum in *a*

5.2. Software

We limit ourselves to a sketch of the aims and accomplishments to date of our research on Ultracomputer systems software.

From the beginning of this work, we have focused on the question of how friendly the machine would prove for the ordinary scientific

[4]Note that a shared read-write variable may, for a portion of its lifetime, be used only by one processor or may be read-only. During that time period, the variable may be cached.

programmer. We consider it important that the Ultracomputer hardware and software should make it possible to transport existing major software applications without complete revision, and that radically new languages, programming styles, or user interfaces be avoided where unnecessary[5]. Another highly desirable property for an operating system is the ability to simultaneously handle jobs of all sizes, executing for many users. In particular, the system must provide rapid turnaround for small jobs, necessary during program development. In addition, since the full complement of Ultracomputer processors may only be usable efficiently on very large problems, economic use of the machine requires that a subset of the available processors be readily available for work on small or medium scale problems.

5.2.1. Simulators, Languages, and Programming Style A very early tool in our research was a simulator in which a CDC 6600 was programmed to act like an ideal highly parallel computer with 6600's as the individual processors. This was accomplished by interpreting many parallel instruction streams in round robin fashion, with pointers to the private memory for each processor and has allowed existing FORTRAN compilers to be used without modification. A few new instructions were added, notably Fetch and Add, and library routines were written to make them available in FORTRAN.

Work with this simulator allowed us to explore three styles of programming, all of which will be significant for realization of scientific applications to be run on a full-scale Ultracomputer.

5.2.2. Operating Systems for Highly Parallel MIMD Computers We have developed critical-section-free Ultracomputer implementations for many important algorithms that appear in both applications and systems programs, including queue management, handling of readers-writers, and so forth [9]. We believe that the Ultracomputer, with its ability to combine Fetch and Adds, is the only proposed architecture that enables such algorithms to be free of serial bottlenecks.

An operating system in which jobs can spawn cooperating tasks does not differ much from one that can support multiprogramming, at least at a conceptual level. However, for a machine like the Ultracomputer in which there may be thousands of PEs and many thousands of tasks active or awaiting the attention of a processor, it is very important that there be no centralized monitor or operating system kernel residing in a single processor, since this would necessarily create serious contention for that processor. The result would be a bottleneck preventing high-speed creation of new tasks and

[5] We do not contend that current programming environments are ideal or even adequate. We believe, however, that the ability to support both old and new environments is essential.

fast task assignment to idle processors. Our software designs eliminate this problem by allowing each processor to carry out the functions of the operating system when required. All idle processors access a global queue of tasks awaiting attention. To spawn subtasks or create entirely new jobs, a processor enters appropriate items in the same global queue. On the face of it, this would only seem to move the difficulty of bottlenecks elsewhere, since the standard way of entering and deleting from queues requires that access be serialized (e.g. by a "critical section") to prevent more than one processor from inserting into the same queue slot or from withdrawing the same information. However, we have shown that the properties of the Fetch and Add make it ideal for modifying queue pointers; that is, if F&A(p,1) is used by two processors to increment a queue pointer stored at p, then it is guaranteed that they will obtain different values as required for correct operation. The properties of both network and switches then guarantee that concurrent modifications of queue pointers can take place without delay.

5.3. Pre-verification of the Ultracomputer Design by Study of Applications under Simulation

As indicated above, we have carried out studies in which a number of programs written in FORTRAN were executed under control of an interpreter that made a serial computer act like a parallel computer. Our simulator accumulated statistics from which one could infer the effectiveness of a particular organization of a program and related data. In particular, the total number of instructions, both useful and idle, was counted. The number of references to private and thus cacheable variables relative to references to shared data was also scored to help estimate network traffic.

To see what is implied by such measurements, assume that N processors are dedicated to a given problem so that if any of them must wait for the completion of work by other processors, it can only idle. Since, in principle, such processors could in fact carry out useful work for other jobs, this gives a conservative estimate of efficiency. If a program can be divided among N processors in such a way that each carried out exactly 1/N of the work of a serial program, it could be completed N times as fast as on a single processor. Normally this is not possible: some overhead associated with process creation, synchronization, and idling (because of load imbalance or awaiting intermediate results from another processor) will always occur. Let C(N) be the average number of instructions per processor when N are used. Attained efficiency, E, is the ratio of the instructions per processor for an ideal perfectly parallel algorithm, C(1)/N, to C(N). That is,

$$E = \frac{C(1)}{NC(N)}$$

It is not surprising that if $C(1) << N$, it is impossible to find an efficient parallel decomposition, since at least $N - C(1)$ processors will have nothing

to do, to say nothing of the organizational overhead required. In other words, efficient use of large parallel systems only becomes possible when the problem running on it is large enough to warrant the use of N processors. There is, of course, no lack of such problems, but it is important to bear this point in mind when attempting to predict the efficiency with which it will prove possible to use very large parallel machines.

For sufficiently large problems, all our simulations have shown that efficient parallel programs could be created. For example, while 256 processors cannot be used very effectively to transform a 32x32 matrix to tridiagonal form, the efficiency being only 25%, they can be used with 96% efficiency for a 256x256 matrix. In the same way large problems of three-dimensional fluid flow were found to be handled efficiently by 4096 processors. Note that we have simulated the use of large parallel computers for a variety of problems ranging from transport Monte Carlo to incompressible fluid flow to matrix transformation algorithms. Our studies involved both new programs and adaptation of existing ones, some of the latter poorly written and documented. In every case, the difference from conventional FORTRAN on a sequential machine required only fairly simple, usually high-level reorganization that differed from program to program and that had to be recognized in advance by the programmer.

These conclusions, confirmed both by our earliest work (1967) with simulated large-scale parallel computers, and by more extensive recent simulations, are reconfirmed by the increasing mass of experience currently being generated by LASL and Argonne work with the HEP MIMD parallel computer emulator. There is emerging a clear consensus that large MIMD systems like the Ultracomputer can be expected to perform at over 50% of peak performance for a very wide range fo large-scale scientific computations.

The studies referenced here do not include the effects of network latency and contention. Assuming that all instructions and private data are cached, and using an approximate stochastic model [10] of the network (later found to be accurate until the network is nearly saturated) [11], Snir [12] investigated the reduction in efficiency that results. In four important numerical problems he concluded that the effect was a loss of about 30%. In work partially conducted at IBM, McAuliffe [13] has determined from detailed simulations of the caches and of the network that Snir's result is substantially correct. Further improvement is expected when a "prefetching" policy becomes possible and when temporarily read-only data is cached. We expect that properly designed caches and associated compilers will serve well to conceal the effect of network latency.

6. Present Status Of The Ultracomputer Project

Our simulator studies have led us to conclude that the Ultracomputer has great promise as a highly parallel general purpose architecture. Distributed coordination is necessary both for efficient load balancing in most heterogeneous applications including many Monte Carlo algorithms, and for supporting a decentralized multiprogramming operating system. Effective caches can be designed. Our studies of diverse applications under simulation indicate that effective use of the aggregate hardware will not be be difficult on the large computations now awaiting the necessary resources and the even larger ones that will arise in the future. The combination of MIMD with shared memory leads to a computation model likely to be easily programmable: novel languages will not be required; no particular programming style nor algorithmic constraints are imposed upon the user. As we have seen, Monte Carlo algorithms come in many flavors and require different styles for different problems and methods.

It seems likely that a rather wide range of hardware providing a corresponding range of performance can be built following the architectural ideas outlined here. That would mean that machines ranging from a uniprocessor, through a desk top machine with 64 processors and having performance comparable to the supercomputers of today, and culminating with a serious machine with perhaps 16384 processors capable of delivering sustained performance of about 150 billion operations per second would be possible. All would use common software and would look identical to the user except for speed and memory size.

Because it requires few basic although substantial VLSI components, this architecture would migrate smoothly as technology advanced.

It promises to play a special role in the future of Monte Carlo calculations. Since the full power of its parallelism will be delivered to those algorithms, it will enhance Monte Carlo as the method of choice in many areas. In addition, it promises to provide the efficacy and flexibility needed to explore improved algorithms and new applications.

At present our effort has advanced to include the construction of two prototypes based on Motorola 68010 processors, one having eight, the other four processors. Caches have been constructed and are operating along with the processors.

Perhaps even more important is the fact that operating systems based upon UNIX (tm ATT Bell Laboratories) have also been brought into operation. The migration to the kind of fully parallel decentralized organization sketched above is under way.

Finally, and probably most significant, is the fact that a group of researchers at IBM's Yorktown Heights Laboratory, including G.F. Pfister and S.L. Harvey, has made intensive studies of the architecture, proposed important enhancements, and are now designing a prototype having many processors.

REFERENCES

[1] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.M. Teller, and E. Teller. *J. Chem. Phys.* Vol. 21, p. 1087. 1983.

[2] S. Wansleben, J.G. Zabolitzky, and C. Kalle. *J. Stat. Phys.*, Vol. 37, nos. 3-4, p. 271. November, 1984.

[3] G.O. Williams and M.H. Kalos. *J. Stat. Phys.*, Vol. 37, nos. 3-4, p. 283. November, 1984.

[4] K. Schmidt. *Phys. Rev. Letters*, Vol. 51, p. 2175. December 12, 1983.

[5] A.B. Bortz, M.H. Kalos, and J.L. Lebowitz. *J. Comp. Phys.*, Vol. 17, p. 10. 1975.

[6] A. Gottlieb and J.T. Schwartz. "Networks and Algorithms for Very Large Scale Parallel Computations," *Computer*. January, 1982.

[7] L.J. Guibas and F.M. Liang. "Systolic Stacks, Queues, and Counters", in *Proc. Conf. Advanced Research VLSI*. January, 1982.

[8] M. Snir and J.A. Solworth. "The Ultraswitch – Designing a VLSI Network Node for Parallel Processing", Ultracomputer Note 39, Courant Institute, NYU. 1984.

[9] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. "Basic Techniques for the Efficient Coordination of Very Large Numbers of Coordinating Sequential Processors", *ACM Toplas*. pp. 164-189. April, 1983.

[10] C. Kruskal and M. Snir, "Some Results on Multistage Interconnection Networks for Multiprocessors", Ultracomputer Note 41, Courant Institute, NYU. 1983.

[11] G. Pfister, private comments. 1983.

[12] M. Snir. "NETSIM Network Simulator for the Ultracomputer", Ultracomputer Note 28, Courant Institute, NYU. 1981.

[13] K.P. McAuliffe, Dissertation, NYU. 1984.

ULTRACOMPUTER NOTE #84

HAS BEEN SUPERCEDED BY

ULTRACOMPUTER NOTE #91